HPC Tools

Task 3

Unai Iborra Albéniz

https://github.com/UNIX-73/hpctools (this document is also inside the repo as TASK3.pdf)

After delivering the Task2, I realised that the benchmark results were executed without using the node in an exclusive state, in order to avoid other users using the same node from adding overhead. Because of this I ran the benchmark again with an exclusive node. I also tested with clang, icc and icx compilers.

The new results with an exclusive node are the following (median of 5 iterations in seconds):

=== clang ===

Matrix Size	O0	O1	O2	O3	Ofast	Ref
1024×1024	7.709	2.095	1.971	1.775	1.809	0.131
2048×2048	68.404	20.615	20.238	18.854	18.092	0.761
4096×4096	880.336	359.312	343.268	329.89	340.305	4.39

=== icx ===

Matrix Size	O0	O1	O2	O3	Ofast	Ref
1024×1024	7.615	1.902	1.98	2.14	2.102	0.067
2048×2048	68.212	20.503	19.824	20.621	21.008	0.442
4096×4096	868.55	348.348	353.292	347.995	346.775	3.027

=== icc ===

Matrix Size	O0	O1	O2	O3	fast	Ref
1024×1024	6.862	2.123	1.925	1.745	1.763	0.067
2048×2048	62.633	20.939	20.484	18.801	18.66	0.442
4096×4096	848.42	355.974	357.836	346.188	344.431	3.026

=== gcc_8_4_0 ===

Matrix Size	O0	O1	O2	O3	Ofast	Ref
1024×1024	6.329	1.991	1.98	1.757	1.831	0.109
2048×2048	58.254	20.419	20.003	18.986	18.543	0.803
4096×4096	800.427	352.312	343.066	334.051	324.401	5.646

=== gcc_10_1_0 ===

Matrix Size	O0	O1	O2	O3	Ofast	Ref
1024×1024	6.326	1.984	2.085	1.938	1.791	0.109
2048×2048	57.584	20.227	20.512	18.935	18.719	0.803
4096×4096	798.013	352.5	346.068	335.678	323.617	5.649

```
                          === gcc_11_4_0 ===
| Matrix Size    |       O0 |       O1 |       O2 |       O3 |    Ofast |    Ref |
|----------------|----------|----------|----------|----------|----------|--------|
| 1024x1024      |    6.299 |    2.052 |    1.963 |    1.788 |    1.833 | 0.131 |
| 2048x2048      |   58.745 |   20.594 |   20.235 |   18.664 |   18.169 | 0.763 |
| 4096x4096      |  833.838 |  350.129 |  343.042 |  335.513 |  332.769 | 4.398 |
```

First, I will compare the previously benchmarked compilers (`gcc_*`):

The results when executing in an exclusive node seem more consistent between the compilers but it is unexpectedly slower in 4096x4096 some executions with good optimization levels. The executed binaries are exactly the same as the task 2 binaries so it seems quite strange that for example in gcc-10 Ofast 4096, the result without the exclusive node was 257.9s while it is 323.6s with the exclusive node.

I do not have an explanation for the performance change between the not exclusive and exclusive benchmark. The big differences between results are not present in many instances, most make sense and are faster in exclusive, but still seems strange for the ones that differ from the expected.

icc, icx and clang results:

Clang seems to perform very similarlly to gcc, however, O3 seems to be faster for clang than Ofast for this code. Maybe the optimizations it made weren't as good as the compiler thought and less optimization in this case is faster. Some executions are faster in gcc while others are better in clang so It doesn't seem worse than gcc.

Icc and icx seem to perform slightly worse than all the other compilers. It seems to be a quite small difference. I find this strange as they should be very optimized for intel architectures. The used dgesv implementation for the reference, seems to be quite faster than the default lapacke one.

Manual vectorization

Afer not being able to get full vectorization of the code in Task 2 (only partial vectorization according to the logs), I attempted to manually vectorize the code in order to check if I could get better performance than the autovectorizated code. The code for the manually vectorized code is inside /src_vec_manual/*. In order to get the code vectorized, the avx_double.h header was made. It consists of the following macros wrappping the #include <immintrin.h> functions:

```c
#if defined(__AVX512F__)

typedef __m512d double_vec;
#define VEC_DOUBLE_LEN 8
#define VEC_BYTES 64
#define VEC_NAME "AVX-512"

#define vec_loadu(p) _mm512_loadu_pd(p)
#define vec_storeu(p, v) _mm512_storeu_pd(p, v)
#define vec_set1(x) _mm512_set1_pd(x)

#define vec_add(a, b) _mm512_add_pd(a, b)
```

```
#define vec_sub(a, b) _mm512_sub_pd(a, b)
#define vec_mul(a, b) _mm512_mul_pd(a, b)
#define vec_div(a, b) _mm512_div_pd(a, b)
#define vec_fmadd(a, b, c) _mm512_fmadd_pd(a, b, c)

#elif defined(__AVX2__) ...
```

They allow to make the code more portable for different architectures, as it uses macro defines to wrap the different SIMD instructions. This way the vectorized code can be adapted easily to the architecture of the machine. For example, my personal laptop uses SSE2, while the FT3 uses AVX512, but as the functions are used with macros, it works for my laptop architecture and the FT3 architecture (as long as it is compiled with -march=native or the correct -march flag for each machine).

The first part of the gauss implementation was succesfully vectorized. The general idea of manual vectorization was considerin when the number of remaining iterations allow to execute the vectorized instructions, or as the iteration number is smaller than the vector size it must be performed without vectorization.
For example:

```
for (size_t row = col + 1; row < n; row++) {
double piv2 = a[row][col];
double mul = piv2 / piv1;

// a
for (size_t i = col; i < n; i++) a[row][i] -= mul * a[col][i];

...
```

Was vectorized as:

```
for (size_t row = col + 1; row < n; row++) {
      double piv2 = a[row][col];
      double mul = piv2 / piv1;

      double_vec mul_vec = vec_set1(mul);

      // a
      double *a_col_ptr = &a[col][col];
      double *a_row_ptr = &a[row][col];

      size_t i = col;

      for (; i + VEC_DOUBLE_LEN <= n; i += VEC_DOUBLE_LEN) {
            // mul_result = mul * a[col][i]
            double_vec a_col_vec = vec_loadu(a_col_ptr);
            double_vec mul_result = vec_mul(mul_vec, a_col_vec);

            // sub_result = a[row][i] - mul_result
            double_vec a_row_vec = vec_loadu(a_row_ptr);
            double_vec sub_result = vec_sub(a_row_vec, mul_result);

            // store value on matrix
            vec_storeu(a_row_ptr, sub_result);

            a_col_ptr += VEC_DOUBLE_LEN;
            a_row_ptr += VEC_DOUBLE_LEN;
      }
      // standard scalar operations for remaining values
      for (; i < n; i++) {
            *a_row_ptr -= mul * (*a_col_ptr);

            a_col_ptr++;
            a_row_ptr++;
    }
```

Other parts of the dgesv function were also manually vectorized, watch /src_vec_manual/dgesv.c in order to watch the implementation.

Before manually vectorizing the second part of the gauss function (resolve_triangle_matrix / backward sustitution), as it was harder to vectorize, in order to verify the manual implementation improved the performance, a benchmark was ran. The results from the benchmark are the following:

=== gcc_8_4_0 ===

Matrix Size	O0	O1	O2	O3	Ofast	Ref
1024×1024	5.38	1.855	1.874	1.852	1.753	0.109
2048×2048	50.629	18.695	18.737	18.921	18.505	0.805
4096×4096	742.197	350.811	339.81	333.157	339.451	5.652

=== gcc_10_1_0 ===

Matrix Size	O0	O1	O2	O3	Ofast	Ref
1024×1024	5.395	1.756	1.98	1.763	1.785	0.109
2048×2048	51.065	18.693	19.192	18.892	18.869	0.806
4096×4096	763.312	351.06	344.822	341.948	331.883	5.652

=== gcc_11_4_0 ===

Matrix Size	O0	O1	O2	O3	Ofast	Ref
1024×1024	5.383	1.879	1.775	1.776	1.77	0.132
2048×2048	50.373	19.445	18.928	18.725	18.549	0.764
4096×4096	763.743	348.537	341.042	340.288	325.83	4.398

=== icc ===

Matrix Size	O0	O1	O2	O3	fast[1]	Ref
1024×1024	5.692	1.921	1.823	1.816	1.819	0.062
2048×2048	52.855	19.354	19.319	19.33	18.976	0.391
4096×4096	767.005	340.737	353.717	351.644	360.228	2.775

=== icx ===

Matrix Size	O0	O1	O2	O3	Ofast	Ref
1024×1024	6.218	1.805	1.791	1.883	1.912	0.062
2048×2048	57.09	18.856	18.572	18.33	19.133	0.391
4096×4096	790.609	350.257	333.12	345.101	342.577	2.78

=== clang ===

Matrix Size	O0	O1	O2	O3	Ofast	Ref
1024×1024	6.2	1.948	1.851	1.77	1.858	0.132
2048×2048	57.101	19.037	19.36	19.712	18.617	0.764

1 Although the python scripts compiled with -Ofast, I manually compiled the binary with -fast, so they are the correct results.

```
 | 4096×4096     | 784.618 | 337.737 | 332.522 | 345.647 | 339.952 | 4.392 |
```

Comparing the results to the previous benchmark, the results are similar in most cases, so I decided to stop manually vectorizing. For now onwards the used code was the one inside /src_vec/.

Heap usage

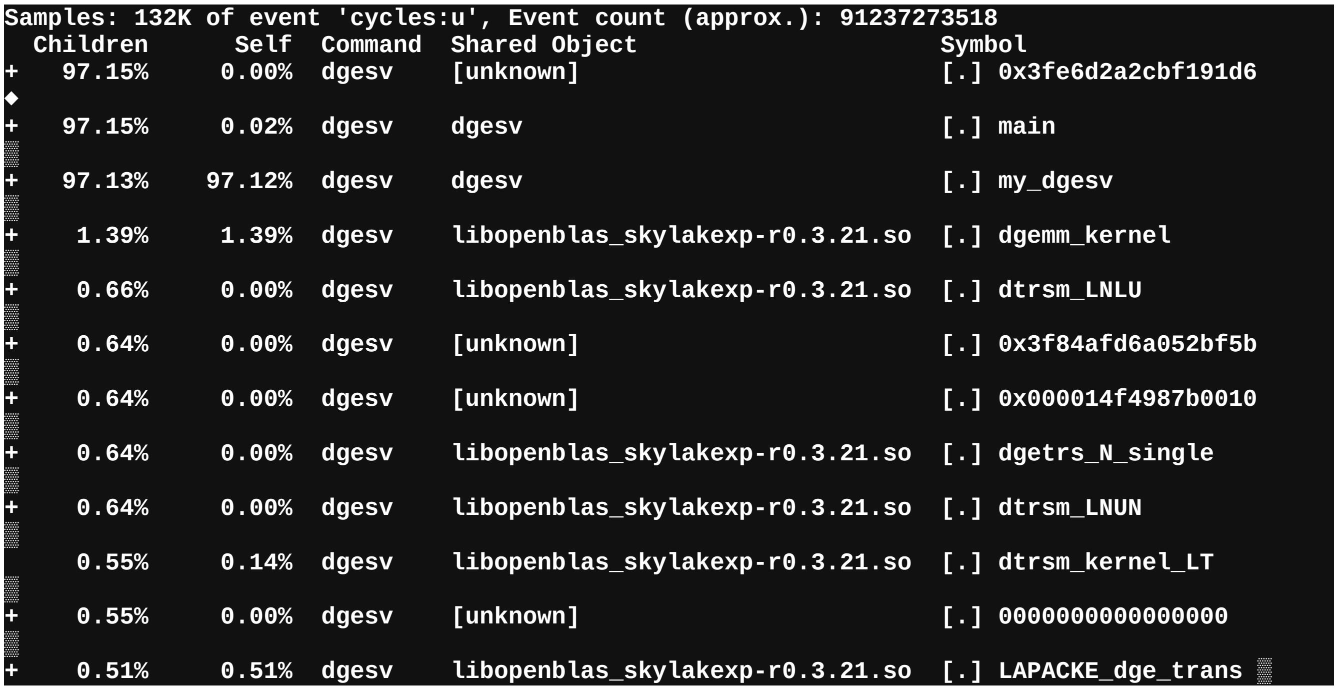
The developed dgesv function does not use heap allocations nor performs any frees. The provided main code for the task exercise although does not free the three allocations it performs. For the current state of the code, not freeing those allocations is not a critial error, as after executing the lapacke dgesv and the custom dgesv it exits and the os deallocates the mmu pages. However, in order to have cleaner and more scalable code, frees have been added.

The use of tools for this specific code to analyze memory leaks or allocation sizes does not seem necessary, as it allocates the needed space, and only three allocations are made during the whole execution of the program.

Cache performance

Perf

In order to analyze the performance of the code, the perf tool was used. The analyzed code was compiled with gcc11.4.0 in the -Ofast configuration and with -ggdb. The results from perf record and report were the following:



```
Samples: 132K of event 'cycles:u', Event count (approx.): 91237273518
  Children      Self  Command  Shared Object                     Symbol
+   97.15%     0.00%  dgesv    [unknown]                         [.] 0x3fe6d2a2cbf191d6
◆
+   97.15%     0.02%  dgesv    dgesv                             [.] main
▒
+   97.13%    97.12%  dgesv    dgesv                             [.] my_dgesv
▒
+    1.39%     1.39%  dgesv    libopenblas_skylakexp-r0.3.21.so  [.] dgemm_kernel
▒
+    0.66%     0.00%  dgesv    libopenblas_skylakexp-r0.3.21.so  [.] dtrsm_LNLU
▒
+    0.64%     0.00%  dgesv    [unknown]                         [.] 0x3f84afd6a052bf5b
▒
+    0.64%     0.00%  dgesv    [unknown]                         [.] 0x000014f4987b0010
▒
+    0.64%     0.00%  dgesv    libopenblas_skylakexp-r0.3.21.so  [.] dgetrs_N_single
▒
+    0.64%     0.00%  dgesv    libopenblas_skylakexp-r0.3.21.so  [.] dtrsm_LNUN
▒
     0.55%     0.14%  dgesv    libopenblas_skylakexp-r0.3.21.so  [.] dtrsm_kernel_LT
▒
+    0.55%     0.00%  dgesv    [unknown]                         [.] 0000000000000000
▒
+    0.51%     0.51%  dgesv    libopenblas_skylakexp-r0.3.21.so  [.] LAPACKE_dge_trans ▒
```

Thre results show that the my_dgesv function takes 97.13% of the computing of the whole execution. This makes sense with the results shown in the benchmarks.

In order to analyze the code in a more in depth way, the *perf stat -e cycles,instructions,branches,branch-misses,L1-dcache-loads,L1-dcache-load-misses,LLC-loads,LLC-load-misses,task-clock ./dgesv 2048* command was executed. It analyzes the cpu usage, memory stalls and efficiency. The results were the following:

```
Performance counter stats for './dgesv 2048':

    87,137,766,555      cycles:u                  #    2.788 GHz
    36,593,372,049      instructions:u            #    0.42  insn per cycle
     4,347,177,451      branches:u                #  139.069 M/sec
         8,242,611      branch-misses:u           #    0.19% of all branches
    11,410,934,756      L1-dcache-loads:u         #  365.044 M/sec
     6,172,943,499      L1-dcache-load-misses:u   #   54.10% of all L1-dcache accesses
     4,289,634,445      LLC-loads:u               #  137.228 M/sec
     1,654,850,305      LLC-load-misses:u         #   38.58% of all LL-cache accesses
         31,259.11 msec task-clock:u              #    1.000 CPUs utilized

      31.266076453 seconds time elapsed

      31.172360000 seconds user
       0.030936000 seconds sys
```

Those results show a result of 0.42 instructions per cycle (87,137,766,555 / 36,593,372,049), which is a quite bad result.

For cache hits and misses, the L1 miss percentage was 54.10%, another quite bad result for the code. Even the last level cache had a 38.58% of misses, which should explain a big part of the bad performance of the code. 38.58% of reads go to main memory instead of from cache. This means the code is very memory bound and should be one of the main points to try to optimize.

Another metric analyzed in a different execution was the branch misses. The results were the following:

```
8,240,805      branch-misses:u           #    0.19% of all branches
```

It shows that the branch misses doesn’t seem to be the causing problem of the bad performance. This is consistent with the code.

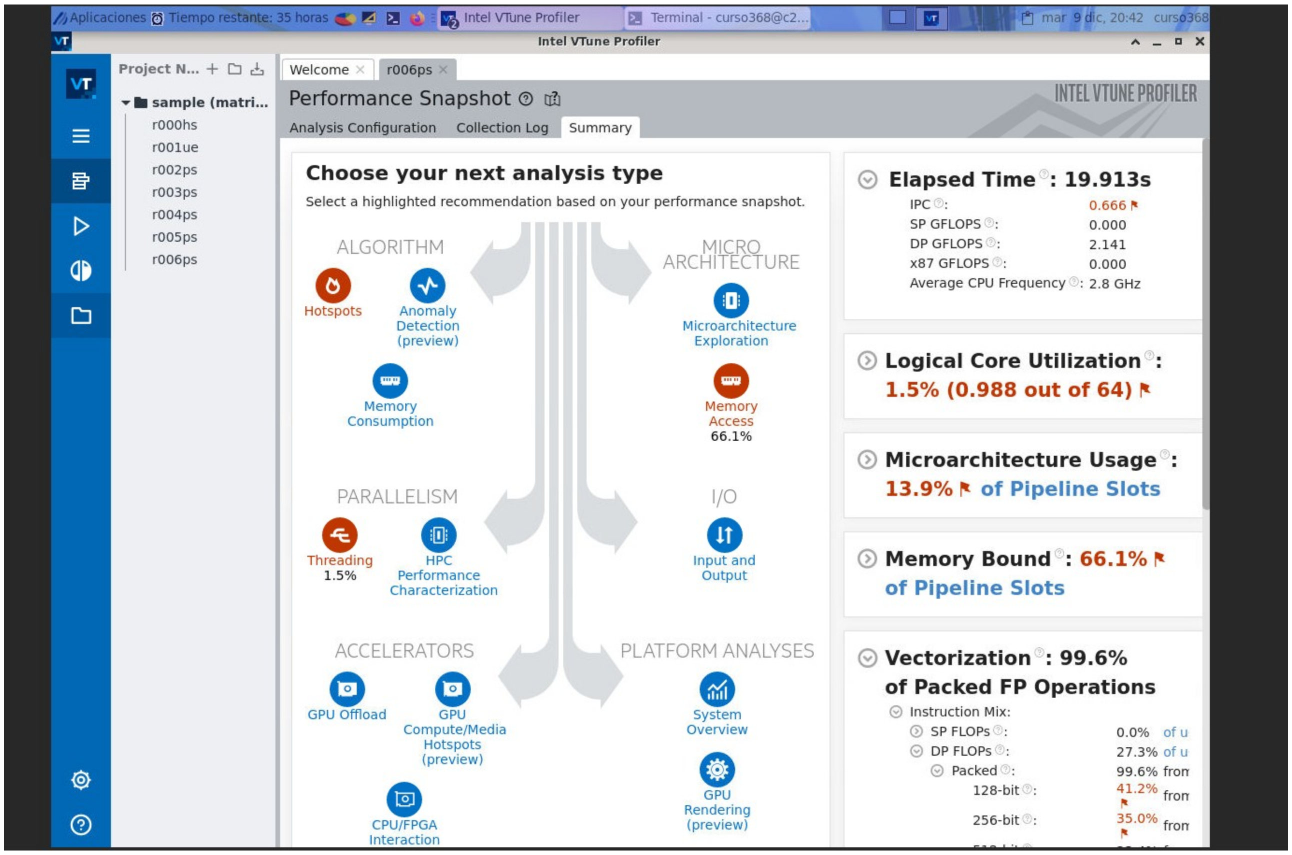
Analyzing the FLOPS and vector instruction performance was not succesfully analyzed. When attempted to use params like fp_arith_inst_retired.512b_packed_double an error was received. After using perf list and not finding those parameters, the asumption that it is not supported param in FT3 or that it requires a module was made.

Gprof

The use of gprof was not seen as needed, as the my_dgesv function only has one function call once to resolve_triangle_matrix (the backwards sustitution).

Intel Vtune Profiler

This tool was used to get a more accurate analysis of the performance of the code. The results of a 4096 size matrix with gcc-11 is the following:



Performance Snapshot

Analysis Configuration | Collection Log | Summary

Choose your next analysis type

Select a highlighted recommendation based on your performance snapshot.

Elapsed Time: 19.913s

IPC:	0.666
SP GFLOPS:	0.000
DP GFLOPS:	2.141
x87 GFLOPS:	0.000
Average CPU Frequency:	2.8 GHz

Logical Core Utilization: 1.5% (0.988 out of 64)

Microarchitecture Usage: 13.9% of Pipeline Slots

Memory Bound: 66.1% of Pipeline Slots

Vectorization: 99.6% of Packed FP Operations

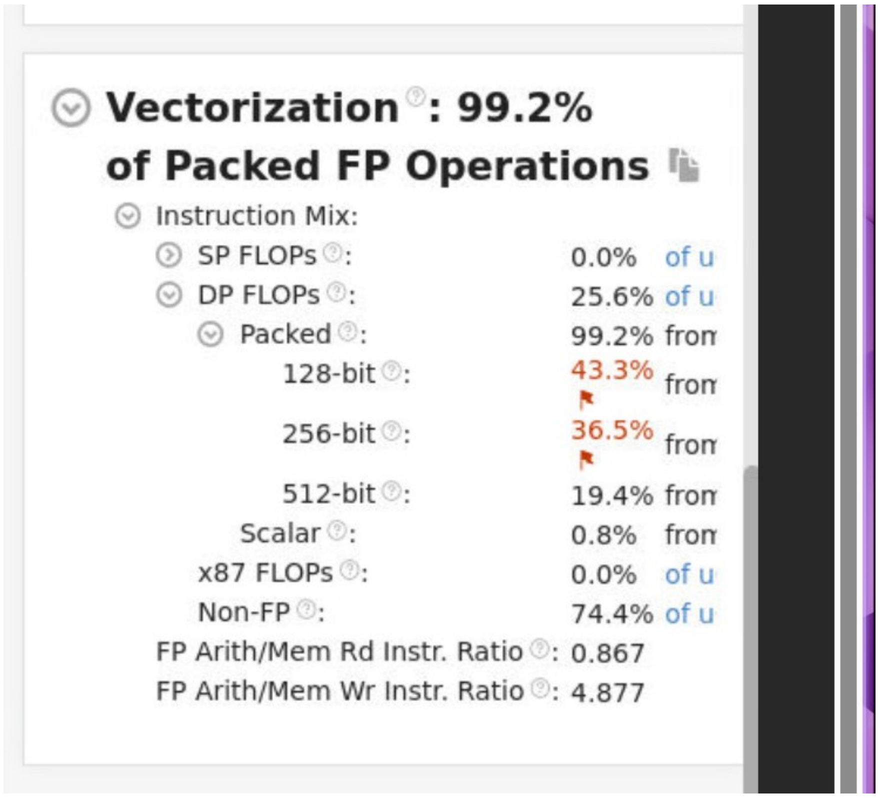
Instruction Mix:		
SP FLOPs:	0.0%	of u
DP FLOPs:	27.3%	of u
Packed:	99.6%	fron
128-bit:	41.2%	fron
256-bit:	35.0%	fron

As analyzed with Perf, the main problem of the program is that it is extremely memory bound. Another bad result the profiler shows is the low use of the available threads. This occurs because the program is single threaded, and because this is intended, it will not be considered.

The tool shows a instructions per cycle of 0.66, better than the 0.42 obtained with Perf but still a very bad result.

An attempt to run a memory profiling inside Vtune was made but another program was needed (it logged an error that it needed another program) and I couldn’t manage to load it for the FT3.

Even though the code is very memory bound, the vectorization seems to be very optimized, the results show a 99.2% of packed floating point operations. This seems to indicate that no further action is needed in the vectorization optimization effort, the effort made in task2 seems to be enough. However, 512-bit FP simd operations are only 19.4% of the vector instructions used. Maybe fixing the vectorization warning logs (that the loops are partially vectorized, but not fully, according to what I inerpreted from the logs) from Task 2 could improve that metric.

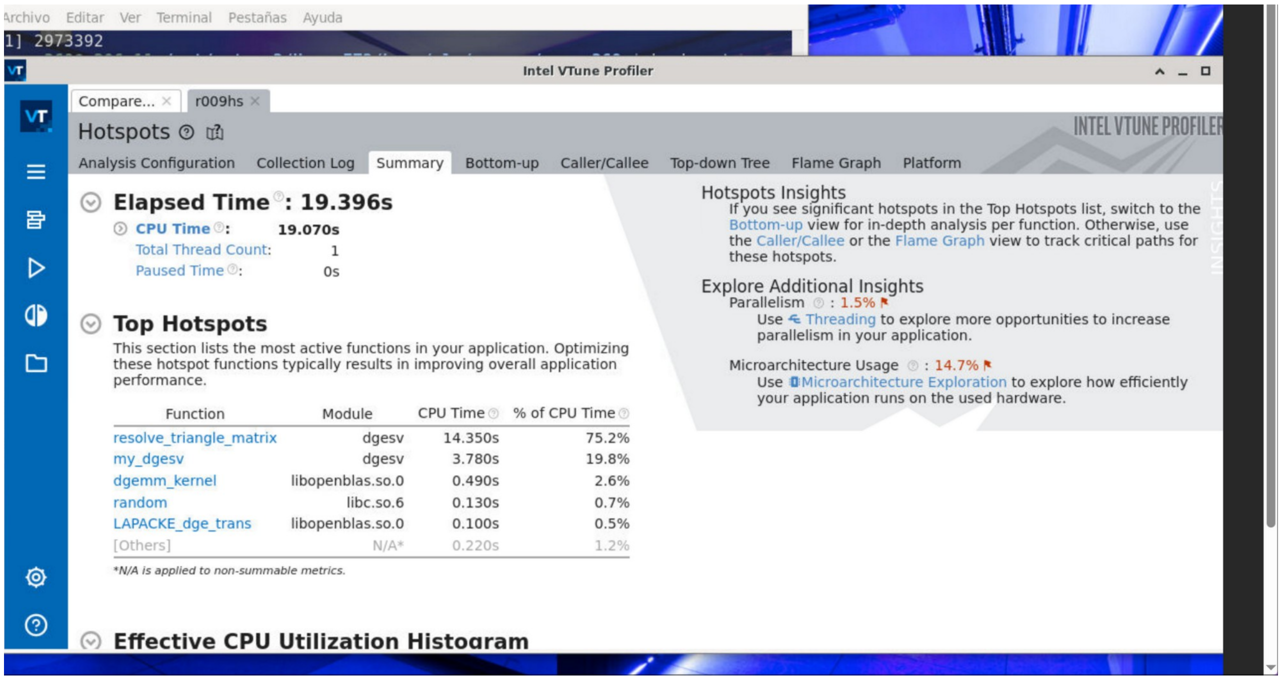


Vectorization: 99.2% of Packed FP Operations

Instruction Mix:

SP FLOPs:	0.0%	of u
DP FLOPs:	25.6%	of u
Packed:	99.2%	fron
128-bit:	43.3%	fron
256-bit:	36.5%	fron
512-bit:	19.4%	fron
Scalar:	0.8%	fron
x87 FLOPs:	0.0%	of u
Non-FP:	74.4%	of u
FP Arith/Mem Rd Instr. Ratio:	0.867	
FP Arith/Mem Wr Instr. Ratio:	4.877	

A hotspots analysis was also made:



```
1] 2973392
```

Hotspots

Elapsed Time: 19.396s

CPU Time:	19.070s
Total Thread Count:	1
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
resolve_triangle_matrix	dgesv	14.350s	75.2%
my_dgesv	dgesv	3.780s	19.8%
dgemm_kernel	libopenblas.so.0	0.490s	2.6%
random	libc.so.6	0.130s	0.7%
LAPACKE_dge_trans	libopenblas.so.0	0.100s	0.5%
[Others]	N/A*	0.220s	1.2%

*N/A is applied to non-summable metrics.

Effective CPU Utilization Histogram

Hotspots Insights

If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee or the Flame Graph view to track critical paths for these hotspots.

Explore Additional Insights

Parallelism : 1.5%

Use Threading to explore more opportunities to increase parallelism in your application.

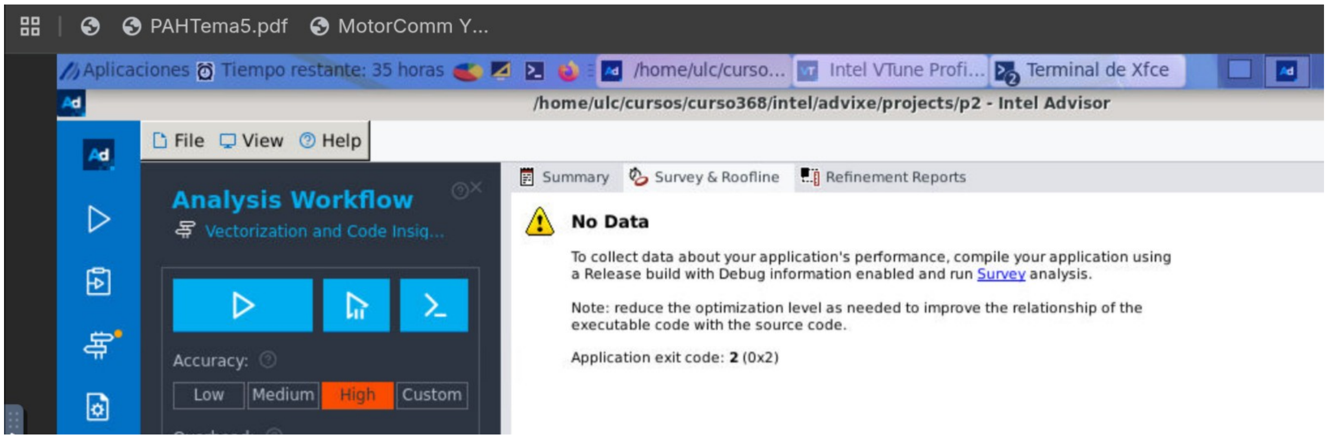
Microarchitecture Usage : 14.7%

Use Microarchitecture Exploration to explore how efficiently your application runs on the used hardware.

This analysis show relevant information that was not seen with Perf. The most relevant function for performance seems to be “resolve_triangle_matrix” (75% of cpu time), the backwards sustitution step of the my_dgesv function. This helps to put the focus on the optimization in that function.

Intel Advisor

I attempted to use Intel Advisor in order to profile more in depth vectorization and more characteristics of the program but the following error message was shown:



/home/ulc/cursos/curso368/intel/advixe/projects/p2 - Intel Advisor

No Data

To collect data about your application's performance, compile your application using a Release build with Debug information enabled and run Survey analysis.

Note: reduce the optimization level as needed to improve the relationship of the executable code with the source code.

Application exit code: **2** (0x2)

Compiling with -g and multiple optimization levels was also attempted but not succesful.

Current status of the optimization

As analyzed with the different tools, the main problem of the code is that it is extremely memory bound, the function that takes more time is the backwards sustitution step and it seems to be vectorized almost fully.

In order to attempt fixing the backwards sustitution function, the main focus must be put in the memory accesses and not breaking the vectorization.

After some time analyzing the code, I believe one of the problems is the following line at the innermost loop:

```
double constant_resolved = b[col][rhs];
```

I belive this is the main problem because rhs is constant inside the inner loop, but col is not. This means that each iteration it is requesting a completely new line (it is making big jumps by accessing like b[idx--][const_idx] instead of a cache friendly b[const_idx][idx++/--]). This error happens more if the n of the matrix is bigger than the required cache for n elements, although a very small matrix already is bigger than the n for a cache line (8 doubles per line) so the bad performance due to bad memory access should be almost present.

I could not manage to improve the code (I made some attempts by trying to access forward inside the inner loop but I did not manage them to work), but im quite confident that one of the main problems of the code is that memory access.